STOCKHOLM UNIVERSITY
Department of Statistics
Ellinor Fackle Fornius

Hand-in assignment 1

R Programming

This document contains R course material designed by Leif Johnsson, Måns Magnusson and Josef Wilzén

Contents

Instructions

The assignments have to be written in `R markdown` with the output in either `pdf` or `HTML` format. Both the `.Rmd` and `.pdf`/`.html`-files have to be handed in. Upload the documents to Athena no later than the deadline for this assignment (stated in the Course Description). Name your files with your group ID and assignment number. The report should include the code as well as the corresponding output. Comment your code when appropriate. For each task in the assignment you should reproduce the example code, when given. Otherwise, construct a relevant example of your own to illustrate the results.

1 `my_ num_vector()`

Create a function called `my_num_vector()` **without** parameters. The function should do the following calculations and return the vector below.

$$\left(\log_{10} 11, \cos\left(\frac{\pi}{5}\right), e^{\pi/3}, (1173 \mod 7)/19\right)$$

In the example below the values have been rounded to fewer decimals. Your functions should return "all" decimals.

```
my_num_vector()

[1] 1.04139 0.80902 2.84965 0.21053
```

2 `mult_first_last()`

Create a function called `mult_first_last()` with the argument `vector`. The function shall return the product of the first and last element in `vector`.

```
mult_first_last(vektor = c(3,1,12,2,4))

[1] 12

mult_first_last(vektor = c(3,1,12))

[1] 36
```

3 `orth_scalar_prod()`

Create a function called `orth_scalar_prod()` which calculate the scalar product between two vectors, `a` and `b`, in an orthonormal base. The scalar product is calculated in the following way:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

```
orth_scalar_prod(a = c(3,1,12,2,4), b = c(1,2,3,4,5))

[1] 69

orth_scalar_prod(a = c(-1, 3), b = c(-3, -1))

[1] 0
```

4 `lukes_father()`

Create a function called `lukes_father()` which takes the argument `name`. The function shall write out `[ name], I am your father.` Where[name] is replaced with the value of the argument `nam` e.

Notice! use `cat()`, not `return()`. **Hint!** the argument `sep` in `cat()`

```
lukes_father(name = "Luke")

Luke, I am your father.

lukes_father(name = "Leia")

Leia, I am your father.
```

5 `approx_e()`

The number e can be defined by the following infinite series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

The number e can be approximated by the following series by choosing an arbitrary value N:

$$e = \sum_{n=0}^{N} \frac{1}{n!}$$

Create a function called `approx_e()` with the argument N to create an arbitrarily approximation of e. Try how large N is needed for the function to approximate e correctly with four decimals.

```
approx_e(N = 2)

[1] 2.5

approx_e(N = 4)

[1] 2.7083
```

6 `filter_my_ vector(x, geq)`

Create a function called `filter_my_vector()` with the arguments `x` and `geq`. The function should take a vector `x` and set all values greater than or equal to `geq` to missing value (`NA`).

See the example below.

```
filter_my_vector(x = c(2, 9, 2, 4, 102), geq = 4)

[1]  2 NA  2 NA NA
```

7 my_magic_matrix()

Create a function called `my_magic_matrix()` without any parameters that creates and returns the following magic matrix.

$$\begin{pmatrix} 4 & 9 & 2 \\ 3 & 5 & 7 \\ 8 & 1 & 6 \end{pmatrix}$$

Can you see what's magic about it?

```
my_magic_matrix()

     [,1] [,2] [,3]
[1,]    4    9    2
[2,]    3    5    7
[3,]    8    1    6
```

8 calculate_elements(A)

Create a function called `calculate_elements(A)` that can take a matrix of an arbitrary size and calculate the number of elements in the matrix.

See examples below:

```
mat <- my_magic_matrix ()
calculate_elements(A = mat)

[1] 9

new_mat <- cbind(mat, mat)
calculate_elements(A = new_mat)

[1] 18
```

9 row_to_zero(A, i)

Create a function called `row_to_zero(A, i)` that can take a matrix of an arbitrary size and set the row indexed with `i` to zero.

See examples below:

```
mat <- my_magic_matrix()
row_to_zero(A = mat, i = 3)
```

```
     [,1] [,2] [,3]
[1,]    4    9    2
[2,]    3    5    7
[3,]    0    0    0

row_to_zero(A = mat, i = 1)

     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    3    5    7
[3,]    8    1    6
```

10 add_elements_to_matrix(A, x, i, j)

Create a function called `add_elements_to_matrix()` with parameters `A`, `x`, `i`, `j`. The function should take a matrix `A` of an arbitrary size and add the value `x` to the parts of `A` indexed by row(s) `i` and column(s) `j`.

See an example below:

```
mat <- my_magic_matrix()
add_elements_to_matrix(A = mat, x = 10, i = 2, j = 3)

     [,1] [,2] [,3]
[1,]    4    9    2
[2,]    3    5   17
[3,]    8    1    6

add_elements_to_matrix(A = mat, x = -2, i = 1: 3, j = 2: 3)

     [,1] [,2] [,3]
[1,]    4    7    0
[2,]    3    3    5
[3,]    8   -1    4
```

Without the last example is the described functionality of the function clear? Are there other possible interpretations when `i` and `j` are vectors? This is to make you think about code documentation.

11 my_magic_list()

Create a function called `my_magic_list()` without parameters that creates and returns a list with three list elements. The first should contain a text element with the text `"my own list"`. The second element should be the vector generated by the function `my_num_vector()` above and the third element should be the matrix generated by `my_magic_matrix()` above.

The first list element should be named `info`.

This is how the list should look.

```
my_magic_list()

$info
[1] "my own list"

[[2]]
[1] 1.04139 0.80902 2.84965 0.21053

[[3]]
     [,1] [,2] [,3]
[1,]    4    9    2
[2,]    3    5    7
[3,]    8    1    6
```

12 change_info(x, text)

Create a function that will take a list `x` (that must contain one element with name `info`) and change this element to the text argument given by `text`.

See an example below:

```
a_list <- my_magic_list()
change_info(x = a_list, text = "Some new info")

$info
[1] "Some new info"

[[2]]
[1] 1.04139 0.80902 2.84965 0.21053

[[3]]
     [,1] [,2] [,3]
[1,]    4    9    2
[2,]    3    5    7
[3,]    8    1    6
```

13 sum_numeric_parts(x)

Create a function called `sum_numeric_parts()` that will take a list `x` and sum together all numeric elements in this list. In a simple implementation you will get warning messages seen below.

```
a_list <- my_magic_list()
sum_numeric_parts(x = a_list)

Warning in sum_numeric_parts(x = a_list): NAs introduced by coercion

[1] 49.911
```

```
sum_numeric_parts(x = a_list [2])

[1] 4.9106
```

14 add_note(x, note)

Create a function that will take a list `x` and add a new list element with the name `note`. This new element should contain text from the `note` parameter.

See an example below:

```
a_list <- my_magic_list()
add_note(x = a_list, note = "This is a magic list!")

$info
[1] "my own list"

[[2]]
[1] 1.04139 0.80902 2.84965 0.21053

[[3]]
     [,1] [,2] [,3]
[1,]    4    9    2
[2,]    3    5    7
[3,]    8    1    6

$note
[1] "This is a magic list!"
```

15 my_data.frame()

Create a function that generates a data.frame that has the following variables and elements.

```
my_data.frame()

  id name income  rich
1  1 John   7.30 FALSE
2  2 Lisa   0.00 FALSE
3  3 Azra  15.21  TRUE
```

16 sort_head(df, var.name, n)

Create a function called `sort_head()` that takes a `data.frame` as parameter `df` and returns the `n` largest values for the given variable `var.name`. All variables should be returned.

See below for an example of the function.

```
data(iris)
sort_head(df = iris, var.name = "Petal.Length", n = 5)
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
119          7.7         2.6          6.9         2.3 virginica
118          7.7         3.8          6.7         2.2 virginica
123          7.7         2.8          6.7         2.0 virginica
106          7.6         3.0          6.6         2.1 virginica
132          7.9         3.8          6.4         2.0 virginica
```

17 add_median_variable(df, j)

Create a function called `add_median_variable()` that should take a `data.frame` and a column id `j`. The function should compute the median for this variable and create a new variable called `compared_to_median` in the `data.frame`. All values that are greater than the median should have the text label `"Greater`”, the values that are smaller should have the label `"Smaller"`. The element that is the median (can happen) should have the label `"Median"`.

Below is an example using the dataset `faithful`.

```
data(faithful)
head(add_median_variable (df = faithful, 1))
```

```
  eruptions waiting compared_to_median
1     3.600      79            Smaller
2     1.800      54            Smaller
3     3.333      74            Smaller
4     2.283      62            Smaller
5     4.533      85            Greater
6     2.883      55            Smaller
```

```
tail(add_median_variable (df = faithful, 2))
```

```
    eruptions waiting compared_to_median
267     4.750      75            Smaller
268     4.117      81            Greater
269     2.150      46            Smaller
270     4.417      90            Greater
271     1.817      46            Smaller
272     4.467      74            Smaller
```

18 analyze_columns(df, j)

Create a function called `analyze_columns()` that should take a `data.frame` called `df` and two column ids in a vector `j` of length 2. These two columns should be analyzed and the results should be returned as a list with three elements. The first two should contained a named vector with the mean, median and the sd for each of the variables. The third element should contain the correlation matrix between the two columns.

The list should be named with the variable names (first two list elements) and the last element should be called `correlation_matrix`. Below are a couple of examples:

```
data(faithful)
analyze_columns(df = faithful, 1:2)

$eruptions
  mean median     sd
3.4878 4.0000 1.1414

$waiting
  mean median     sd
70.897 76.000 13.595

$correlation_matrix
          eruptions waiting
eruptions   1.00000 0.90081
waiting     0.90081 1.00000

data (iris)
analyze_columns(df = iris, c(1,3))

$Sepal.Length
   mean  median      sd
5.84333 5.80000 0.82807

$Petal.Length
  mean median     sd
3.7580 4.3500 1.7653

$correlation_matrix
             Sepal.Length Petal.Length
Sepal.Length      1.00000      0.87175
Petal.Length      0.87175      1.00000

analyze_columns(df = iris, c(4,1))

$Petal.Width
   mean  median      sd
1.19933 1.30000 0.76224

$Sepal.Length
   mean  median      sd
5.84333 5.80000 0.82807

$correlation_matrix
             Petal.Width Sepal.Length
Petal.Width      1.00000      0.81794
Sepal.Length     0.81794      1.00000
```

19 matrix_trace()

Diagonal matrices and the trace of a matrix is important in linear algebra. More information is found here. We will now create a function to calculate the trace of a matrix. The function shall be called `matrix_trace()` and have the argument `X`, which is an arbitrarily large square matrix. The trace of a matrix trace is the sum of its diagonal element:

$$\mathrm{tr}(A) = a_{11} + a_{22} + \cdots + a_{nn} = \sum_{i=1}^{n} a_{ii}$$

The function `matrix_trace()` should return the matrix trace as a numeric value.

Examples of how you can implement the function:

1. Set the values that are not on the diagonal to 0. **Hint!** `upper.tri()`, `lower.tri()`, `diag()`
2. Transform the matrix into a vector and sum the values.

Here are test examples of how the function should work:

```
A <- matrix(2:5, nrow = 2)
matrix_trace(X = A)

[1] 7

B <- matrix(1:9, nrow = 3)
matrix_trace(X = B)

[1] 15

C <- matrix(9:-6, nrow = 4)
C_trace <-matrix_trace(X = C)

C_trace

[1] 6
```

20 fast_stock_analysis()

We will now create a function to do a quick analysis of a dataset which is saved as `.csv`. This is an example of how we can use functions in R. Data sometimes comes in continuously and then we want to do some standard analyses quickly with a pre-programmed function. We will create a function `fast_stock_analysis()` with the argument `file_path` and `period_length`. The purpose is to quickly analyze the share price of recent days.

The function should load a `.csv` file specified by the argument `file_path` and return a list of the named list items `total_spread`, `mean_final`, `final_up` and `dates`.

How to implement the function:

1. First enter a function that load data
2. Load data (csv) with the argument `file_path` (contains both file name and path). Use the function `read.csv()`. **Hint!** `stringsAsFactors` means that the date variables become a factor variable. **Note!** `file_path` should only contain the path to file, the file should be loaded inside the function.

3. Pick the last number of days of `period_length` from the dataset (assume that the latest stock prices are at the top).

4. Calculate the values to be returned by the function:

   (a) `total_spread` (a numeric element) is the difference between the highest value of `High` and the lowest value of `Low` during the period.

   (b) `mean_final` (a numeric element) is the mean of the final price during the period.

   (c) `final_up` is a logical value indicating `TRUE` if the final price on the first day of the period is lower than the final price on the last day of the period.

   (d) `dates` (vector with two text elements) shall contain the first and the last date of the period. **Note!** This should be a text vector, not a factor.

5. Put these values together into a named list with the names above.

Download test data `AppleTest.csv` and `google2.csv` from Athena.

21 `leap_year()`

February 29 is a leap day in the calendar and occurs every fourth years such as 2004, 2008, 2012, 2016 and 2020. That is, the years that are evenly divisible by four. Years that are evenly divisible by 100 contains no leap days if they are not simultaneously divisible by 400. For example the year 1900 did not contain a leap day while the year 2000 contained a leap day.

We will create a function that tests whether a vector of years is leap year or not and returns this as a `data.frame`. Create a function called `leap_year()`. The function should have the argument `years` which should be a text vector.

Examples of how you can implement the function:

1. Convert `years` to a numeric vector.

2. Use the numeric vector to test if each given year is a leap year. Generate a logical vector that is `TRUE` if the year is a leap year. **Hint!** Use modulus operator and relational operators. Create a `data.frame` with two variables, `years` and `leap_year`. The variable years should contain the converted numerical vector `years` and `leap_year` shall contain a logical indicator, `TRUE` if the year is a leap year and otherwise `FALSE`.

Here are a test example of how the function should work:

```
my_test_years1 <- c("1900", "1984", "1997", "2000", "2001")
my_result <- leap_year(years = my_test_years1)
str(my_result)

'data.frame': 5 obs. of  2 variables:
 $ years    : num  1900 1984 1997 2000 2001
 $ leap_year: logi  FALSE TRUE FALSE TRUE FALSE

my_result

  years leap_year
1  1900     FALSE
```

```
2  1984       TRUE
3  1997      FALSE
4  2000       TRUE
5  2001      FALSE
```

```r
my_test_years2 <- c("1988", "2000", "1986", "1901", "2012", "2016", "1200", "1300")
leap_year(years = my_test_years2)
```

```
  years leap_year
1  1988      TRUE
2  2000      TRUE
3  1986     FALSE
4  1901     FALSE
5  2012      TRUE
6  2016      TRUE
7  1200      TRUE
8  1300     FALSE
```